Chapter 6. Discovering SOAP Services

Once a WSDL description of a web service has been created, a service consumer must be able to locate it in order to be able to use it. This is known as *discovery*, the topic of this chapter. In particular, we look at the Universal Description, Discovery, and Integration (UDDI) project and the new Web Services Inspection Language.

WSDL provides a service consumer with all the information they need to interact with a service provider. But how can a consumer learn of services to use? The UDDI project is an industry effort to define a searchable registry of services and their descriptions so that consumers can automatically discover the services they need.

UDDI has two parts: a registry of all a web service's metadata (including a pointer to the WSDL description of a service), and a set of WSDL port type definitions for manipulating and searching that registry.

The latest UDDI specification is Version 2.0. In this book, however, we focus completely on Version 1.0. Version 2.0 has not yet been widely implemented and there is very little support available for it.

UDDI is not the only option for service discovery. IBM and Microsoft have recently announced the Web Services Inspection Language (WS-Inspection), an XML-based language that provides an index of all the web services at a given web location.

The first part of this chapter will focus primarily on UDDI. The last half will briefly introduce WS-Inspection and demonstrate its role inService Discovery.

6.1 The UDDI Registry

The UDDI registry allows a business to publicly list a description of itself and the services it provides. Potential consumers of those services can locate them based on taxonomical information, such as what the service does or what industry the service targets.

The registry itself is defined as a hierarchy of business, service, and binding descriptions expressed in XML.

6.1.1 Business Entity

The business entity structure represents the provider of web services. Within the UDDI registry, this structure contains information about the company itself, including contact information, industry categories, business identifiers, and a list of services provided. Example 6-1 shows a fictitious business's UDDI registry entry.

Example 6-1. A UDDI business entry

```
<businessEntity businessKey="uuid:C0E6D5A8-C446-4f01-99DA-70E212685A40"
                operator="http://www.ibm.com"
                authorizedName="John Doe">
   <name>Acme Company</name>
   <description>
     We create cool Web services
   </description>
   <contacts>
     <contact useType="general info">
       <description>General Information</description>
       <personName>John Doe</personName>
       <phone>(123) 123-1234</phone>
       <email>jdoe@acme.com</email>
     </contact>
   </contacts>
   <businessServices>
      ...
   </businessServices>
   <identifierBag>
      <keyedReference
           TModelKey="UUID:8609C81E-EE1F-4D5A-B202-3EB13AD01823"
           name="D-U-N-S"
           value="123456789" />
   </identifierBag>
   <categoryBag>
      <keyedReference
           TModelKey="UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"
           name="NAICS"
           value="111336" />
   </categoryBag>
</businessEntity>
```

6.1.2 Business Services

The business service structure represents an individual web service provided by the business entity. Its description includes information on how to bind to the web service, what type of web service it is, and what taxonomical categories it belongs to. Example 6-2 show a possible business service structure for the Hello World web service.

Example 6-2. Hello World business structure in UDDI

```
<businessService serviceKey="uuid:D6F1B765-BDB3-4837-828D-8284301E5A2A"
                 businessKey="uuid:C0E6D5A8-C446-4f01-99DA-70E212685A40">
   <name>Hello World Web Service</name>
   <description>A friendly Web service</description>
   <bindingTemplates>
      ...
   </bindingTemplates>
   <categoryBag />
</businessService>
```

Notice the use of the Universally Unique Identifiers (UUIDs) in the `businessKey` and `serviceKey` attributes. Every business entity and business service is uniquely identified in all UDDI registries through the UUID assigned by the registry when the information is first entered.

6.1.3 Binding Templates

Binding templates are the technical descriptions of the web services represented by the business service structure. A single business service may have multiple binding templates. The binding template represents the actual implementation of the web service (it is roughly equivalent to the `service` element we saw in WSDL). Example 6-3 shows a binding template for Hello World.

Example 6-3. A binding template for Hello World

```
<bindingTemplate serviceKey="uuid:D6F1B765-BDB3-4837-828D-8284301E5A2A"
                 bindingKey="uuid:C0E6D5A8-C446-4f01-99DA-70E212685A40">
  <description>Hello World SOAP Binding</description>
  <accessPoint URLType="http">
    http://localhost:8080
  </accessPoint>
  <TModelInstanceDetails>
    <TModelInstanceInfo
           TModelKey="uuid:EB1B645F-CF2F-491f-811A-4868705F5904">
      <instanceDetails>
        <overviewDoc>
          <description>
            references the description of the
            WSDL service definition
          </description>
          <overviewURL>
            http://localhost/helloworld.wsdl
          </overviewURL>
        </overviewDoc>
      </instanceDetails>
    </TModelInstanceInfo>
  </TModelInstanceDetails>
</bindingTemplate>
```

Because a business service may have multiple binding templates, the service may specify different implementations of the same service, each bound to a different set of protocols or a different network address.

6.1.4 TModels

A `TModel` is a way of describing the various business, service, and template structures stored within the UDDI registry. Any abstract concept can be registered within UDDI as a `TModel`. For instance, if you define a new WSDL port type, you can define a `TModel` that represents that port type within UDDI. Then, you can specify that a given business service implements that port type by associating the `TModel` with one of that business service's binding templates.

A `TModel` representing the `HelloWorldInterface` port type looks like Example 6-4.

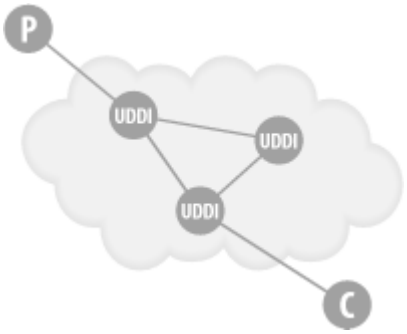
Example 6-4. A TModel for Hello World

```
<TModel TModelKey="uuid:xyz987..."
        operator="http://www.ibm.com"
        authorizedName="John Doe">
  <name>HelloWorldInterface Port Type</name>
  <description>
    An interface for a friendly Web service
  </description>
  <overviewDoc>
    <overviewURL>
      http://localhost/helloworld.wsdl
    </overviewURL>
  </overviewDoc>
</TModel>
```

6.1.5 Federated UDDI Registries

At its core, UDDI is comprised of a global network of linked (*federated*) registries that all implement the same SOAP-based web service interface for publishing and locating web services. Figure 6-1 illustrates this.

Figure 6-1. UDDI registries can be linked to provide a rudimentary distributed search capability



6.1.6 Private UDDI Registries

As an alternative to using the public federated network of UDDI registries available on the Internet, companies or industry groups may choose to implement their own private UDDI registries. These exclusive services would be designed for the sole purpose of allowing members of the company or of the industry group to share and advertise services amongst themselves.

The key to this, however, is that whether the UDDI registry is part of the global federated network or a privately owned and operated registry, the one thing that ties it all together is a common web services API for publishing and locating businesses and services advertised within the UDDI registry.

6.2 The UDDI Interfaces

A registry is no use without some way to access it. The UDDI standard specifies two `SOAP` interfaces for service consumers and service providers to interact with the registry. Service consumers use `InquireSOAP` to find a service, and service providers use `PublishSOAP` to list a service. These services are described with WSDL. The following explanation of the SOAP

APIs refers to the WSDL, but abbreviates some of the repetitive parts. The full WSDL specification of the UDDI API is given in Appendix B.

The core of the UDDI interfaces is the UDDI XML Schema definitions. These define the fundamental UDDI data types, for instance, the `businessDetail`, which communicates detailed information about registered business entities. The UDDI XML Schema must be imported into the WSDL description from its network location at http://www.uddi.org/schema/2001/uddi_v1.xsd, as shown in Example 6-5.

Example 6-5. Importing the WSDL description

```
<import namespace="urn:uddi-org:api"
location="http://www.uddi.org/schema/2001/uddi_v1.xsd"
/>
```

6.2.1 The Publisher Interface

The Publisher interface defines sixteen operations for a service provider managing its entries in the UDDI registry:

`get_authToken`

Retrieves an authorization token. It works exactly like the authorization token we used in the Publisher example in Chapter 3. All of the Publisher interface operations require that a valid authorization token be submitted with the request.

`discard_authToken`

Tells the UDDI registry to no longer accept a given authorization token. This step is equivalent to logging out of the system.

`save_business`

Creates or updates a business entity's information contained in the UDDI registry.

`save_service`

Creates or updates information about the web services that a business entity provides.

`save_binding`

Creates or updates the technical information about a web service's implementation.

`save_TModel`

Creates or updates the registration of abstract concepts managed by the UDDI registry.

`delete_business`

Removes the given business entities from the UDDI registry completely.

`delete_service`

Removes the given web services from the UDDI registry completely.

`delete_binding`

Removes the given web service technical details from the UDDI registry.

`delete_TModel`

Removes the specified `TModel`s from the UDDI registry.

`get_registeredInfo`

Returns a summary of everything the UDDI registry is currently keeping track of for the user, including all businesses, all services, and all `TModel`s.

In the WSDL, these methods correspond to messages based on the underlying UDDI data types, as in Example 6-6.

Example 6-6. UDDI method definition

```
<message name="bindingDetail">
 <part name="body"
       element="uddi:bindingDetail" />
</message>

<message name="businessDetail">
 <part name="body"
       element="uddi:businessDetail" />
</message>
```

The other standard messages are similarly defined.

Finally, we define the port type itself, creating the interface through which modifications can be made to the UDDI registry. Again, only a few definitions have been shown in full detail in Example 6-7, as they all follow the same pattern.

Example 6-7. Representative Publisher operation definitions

```
<portType name="PublishSoap">
  <operation name="delete_binding">
    <input  message="tns:delete_binding" />
    <output message="tns:dispositionReport" />
    <fault  name="error"
            message="tns:dispositionReport" />
  </operation>

  <operation name="delete_business">
    <input  message="tns:delete_business" />
    <output message="tns:dispositionReport" />
    <fault  name="error"
            message="tns:dispositionReport" />
  </operation>
```

```
    <operation name="delete_service">
      <input  message="tns:delete_service" />
      <output message="tns:dispositionReport" />
      <fault  name="error"
              message="tns:dispositionReport" />
    </operation>

    <operation name="delete_TModel"> ...
    <operation name="discard_authToken"> ...
    <operation name="get_authToken"> ...
    <operation name="get_registeredInfo"> ...
    <operation name="save_binding"> ...
    <operation name="save_business"> ...
    <operation name="save_service"> ...
    <operation name="save_TModel"> ...
    <operation name="validate_categorization"> ...
  </portType>
</definitions>
```

6.2.2 The Inquiry Interface

The inquiry interface defines ten operations for searching the UDDI registry and retrieving details about specific registrations:

`find_binding`

> Returns a list of web services that match a particular set of criteria based on the technical binding information.

`find_business`

> Returns a list of business entities that match a particular set of criteria.

`find_ltservice`

> Returns a list of web services that match a particular set of criteria.

`find_TModel`

> Returns a list of `TModel`s that match a particular set of criteria.

`get_bindingDetail`

> Returns the complete registration information for a particular web service binding template.

`get_businessDetail`

> Returns the registration information for a business entity, including all services that entity provides.

`get_businessDetailExt`

Returns the complete registration information for a business entity.

`get_serviceDetail`

Returns the complete registration information for a web service.

`get_TModelDetail`

Returns the complete registration information for a `TModel`.

`InquireSOAP` defines the web service interface for searching the UDDI registry. Example 6-8 shows the method definitions for `find_binding`, `find_business`, and `find_service`.

Example 6-8. InquireSOAP

```
<portType name="InquireSoap">
  <operation name="find_binding">
    <input  message="tns:find_binding" />
    <output message="tns:bindingDetail" />
    <fault  name="error"
            message="tns:dispositionReport" />
  </operation>

  <operation name="find_business">
    <input  message="tns:find_business" />
    <output message="tns:businessList" />
    <fault  name="error"
            message="tns:dispositionReport" />
  </operation>

  <operation name="find_service">
    <input  message="tns:find_service" />
    <output message="tns:serviceList" />
    <fault  name="error"
            message="tns:dispositionReport" />
  </operation>
```

The message definitions are as straightforward as in the Publisher interface. Example 6-9 shows the first three. Consult Appendix C for the full list.

Example 6-9. Inquiry message definitions

```
<message name="authToken">
  <part name="body"
        element="uddi:authToken" />
</message>

<message name="bindingDetail">
  <part name="body"
        element="uddi:bindingDetail" />
</message>
```

```
<message name="businessDetail">
  <part name="body"
        element="uddi:businessDetail" />
</message>
```

6.3 Using UDDI to Publish Services

There are several toolkits, both open and closed source, that provide an implementation of the UDDI Publish and Inquiry interfaces. We'll walk you through using an open source package from IBM called UDDI4J (UDDI for Java). You can download this package from http://oss.software.ibm.com/developerworks/projects/uddi4j.

The steps for using UDDI4J to publish web services are:

1. Register the service provider as a UDDI business entity.
2. Specify the categories and identifiers that apply to your business entity entry.
3. Register the web service as a UDDI business service.
4. Specify the categories that apply to your business service entry.
5. Register the implementation details of your web service, including the network location where the service is deployed.

The UDDI data model lets us do all these steps in a single operation.

6.3.1 Registration Program

A Java program to publish the Hello World service is given in Appendix C. We'll step you through the highlights, which demonstrate how to use the UDDI4J toolkit.

You use UDDI4J through a proxy object, which handles the underlying SOAP encoding and decoding. You should initialize the proxy to the UDDI registry as shown in Example 6-10.

Example 6-10. Initializing the UDDI Proxy

```
UDDIProxy proxy = new UDDIProxy(  );
proxy.setPublishURL("https://www-
3.ibm.com/services/uddi/testregistry/protect/
    publishapi");
```

UDDI4J defines classes for the UDDI data types. They have straightforward accessors, so you prepare the business entity record as in Example 6-11.

Example 6-11. Specifying the business entity

```
BusinessEntity business = new BusinessEntity(  );
business.setName("O'Reilly and Associates");
```

Similarly you can specify the categories and identifiers for this business entity. In Example 6-12, we use a North American Industry Classification System (NAICS) category code of 11194.

Example 6-12. Specifying categories and identifiers for the business entity

```
CategoryBag cbag = new CategoryBag(  );
KeyedReference cat = new KeyedReference(  );
cat.setTModelKey("UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2");
cat.setKeyName("NAICS");
cat.setKeyValue("11194");
cbag.getKeyedReferenceVector(  ).add(cat);
business.setCategoryBag(cbag);
```

In Example 6-13, we prepare the identifiers for the business entity. We specify a Dun and Bradstreet number that may be used to identify the business entity (it's fictitious, but you get the idea). Because you can have more than one identifier for a business, UDDI4J defines an `IdentifierBag` class that holds the individual identifiers.

Example 6-13. Business entity identifiers

```
IdentifierBag ibag = new IdentifierBag(  );
KeyedReference id = new KeyedReference(  );
id.setTModelKey("UUID:8609C81E-EE1F-4D5A-B202-3EB13AD01823");
id.setKeyName("D-U-N-S");
id.setKeyValue("1234567890");
ibag.getKeyedReferenceVector(  ).add(id);
business.setIdentifierBag(ibag);
```

Prepare the business service record as in Example 6-14.

Example 6-14. Initializing the business service record

```
BusinessServices services = new BusinessServices(  );
BusinessService service = new BusinessService(  );
service.setName("Hello World Service");
services.getBusinessServiceVector(  ).add(service);
business.setBusinessServices(services);
```

Example 6-15 shows the initialization of the binding templates. The binding template specifies the protocols implemented by a service and the network location. It is the UDDI equivalent to the WSDL binding and service port definition.

Example 6-15. Initializing the binding templates

```
BindingTemplates bindings = new BindingTemplates(  );
BindingTemplate binding = new BindingTemplate(  );
AccessPoint accessPoint = new AccessPoint(  );
accessPoint.setText("http://localhost:8080");
accessPoint.setURLType("HTTP");
binding.setAccessPoint(accessPoint);
bindings.getBindingTemplateVector(  ).add(binding);
service.setBindingTemplates(bindings);
```

Example 6-16 logs onto the UDDI registry and registers the business entity.

Example 6-16. Registering the business entity

```
AuthToken token = proxy.get_authToken("james", "semaj");
Vector businesses = new Vector(  );
businesses.add(business);
proxy.save_business(token.getAuthInfo().getText(  ), businesses);
```

6.3.2 How to Register

You'll need two things before you can use the registration program:

1. You must have a valid user account with the UDDI registry you choose. You acquire one by registering through the HTML-form interface provided by the specific UDDI registry provider.
2. You must have Apache SOAP Version 2.1 or higher in your Java classpath (UDDI4J uses Apache SOAP). To meet this requirement, make sure that *soap.jar*, *mail.jar*, and *activation.jar* are all in your classpath.

There are three common situations that cause an error registering a service:

1. A company may already exist with the specified name.
2. There may be some problem with the information defined.
3. You might not have proper permissions to perform the requested action.

6.3.3 The SOAP Envelope for the Registration

The SOAP envelope sent to the UDDI registry includes all of the registration information for the business entity, as seen in Example 6-17.

Example 6-17. SOAP envelope for the registration

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <SOAP-ENV:Body>

    <save_business generic="1.0" xmlns="urn:uddi-org:api">
      <authInfo>test</authInfo>
      <businessEntity>
        <name>O'Reilly and Associates</name>
        <businessServices>
         <businessService>
          <name>Hello World Service</name>
          <bindingTemplates>
           <bindingTemplate>
             <accessPoint
                  urlType="HTTP">http://localhost:8080</accessPoint>
           </bindingTemplate>
          </bindingTemplates>
         </businessService>
        </businessServices>
```

```
        <identifierBag>
         <keyedReference keyName="D-U-N-S"
                keyValue="1234567890"
                TModelKey="UUID:8609C81E-EE1F-4D5A-B202-3EB13AD01823"/>
        </identifierBag>
        <categoryBag>
         <keyedReference keyName="NAICS"
                keyValue="11194"
                TModelKey="UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"/>
        </categoryBag>
      </businessEntity>
   </save_business>
</SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

6.3.4 Other Issues

Operations like `save_business` are destructive. In other words, when you tell the UDDI registry to save a business entity, the registry will use the information you provide to replace all other information on that business entity that exists in the registry. There are two ways around this:

1. Retrieve the complete business entity record from the UDDI registry prior to making any changes to the information (e.g., publishing a new service). Make all changes directly to the record received from the registry. Saving stores your modified record.
2. Save only the specific parts you are changing. For example, if you already have a business entity registration at a UDDI registry, and all you want to do is register a new service, then you should use the `save_service` operation rather than `save_business`. This cuts down on the amount of data being moved around and compartmentalizes the changes being made.

Example 6-18 uses `save_service` to localize changes.

Example 6-18. Changing only some fields in the registry

```
// Initialize the proxy to the UDDI registry
UDDIProxy proxy = new UDDIProxy(  );
proxy.setPublishURL("https://www-
3.ibm.com/services/uddi/testregistry/protect/publishapi");

// Prepare the business service record
BusinessServices services = new BusinessServices(  );
BusinessService service = new BusinessService(  );
service.setBusinessKey("uuid:C0E6D5A8-C446-4f01-99DA-70E212685A40");
service.setName("Hello World Service");
services.getBusinessServiceVector(  ).add(service);

// Prepare the binding templates
BindingTemplates bindings = new BindingTemplates(  );
BindingTemplate binding = new BindingTemplate(  );
AccessPoint accessPoint = new AccessPoint(  );
accessPoint.setText("http://localhost:8080");
accessPoint.setURLType("HTTP");
binding.setAccessPoint(accessPoint);
bindings.getBindingTemplateVector(  ).add(binding);
service.setBindingTemplates(bindings);
```

```
// Logon to UDDI registry and register
AuthToken token = proxy.get_authToken("username", "password");
Vector services = new Vector(  );
services.add(service);
proxy.save_service(token.getAuthInfo().getText(  ), services);
```

The only real difference is the absence of the business entity and the addition of the `service.setBusinessKey` line. This tells the UDDI registry which business entity to update. This ID is generated automatically by the UDDI registry and returned to the client when the business entity registration is created.

6.4 Using UDDI to Locate Services

UDDI4J can also be used to locate services that have been published within a UDDI registry. The process involves the use of several `find` operations such as `find_business`, `find_service`, and `find_binding`.

Appendix C has a program to search for a business entity and navigate the results of that operation to find out information about the services provided by that entity. We'll discuss the highlights.

`FindQualifiers` modifies the search operations by indicating whether case-sensitive searching is required, whether the results should be sorted ascending or descending, and whether exact name matching is required. The last argument in all of the find operations is the maximum number of results to return. Passing in the number `zero` indicates that all matching results should be returned. Example 6-19 sets up `FindQualifiers` to look for the O'Reilly business.

Example 6-19. FindQualifiers to look for O'Reilly

```
FindQualifiers fqs = new FindQualifiers(  );
FindQualifier fq = new FindQualifier(  );
fq.setText(FindQualifier.sortByNameAsc);
BusinessList list = proxy.find_business("O'Reilly", fqs, 0);
```

Matching business entities are returned along with a listing of the services offered. The listing includes the name and unique identifier of the service. Use the UUID to drill down and get more information about the service, as in Example 6-20.

Example 6-20. Fetching more information about the service

```
BusinessInfos infos = list.getBusinessInfos(  );
for  (Iterator  i  =  infos.getBusinessInfoVector().iterator(   );  i.hasNext(
);;)  {
    BusinessInfo info = (BusinessInfo)i.next(  );
    System.out.println("Business name: " + info.getName(  ));
  for                    (Iterator                    j                    =
info.getServiceInfos().getServiceInfoVector().iterator();
       j.hasNext();;)  {
        ServiceInfo sinfo = (ServiceInfo)j.next(  );
        System.out.println("\tService name: " + sinfo.getName(  ));
    }
}
```

To retrieve more specific information about a given service, use the `get_serviceDetail` operation and pass in the unique identifier of the service you are requesting:

```
ServiceDetail detail = proxy.get_serviceDetail(serviceKey);
```

Using the information contained in the service detail, a client can connect to and invoke the web service.

6.5 Generating UDDI from WSDL

Because there are some variances and overlapping in how WSDL and UDDI support the description of web services, the industry coalition that is driving UDDI has released a document describing the best practices to follow when using UDDI and WSDL together to enable dynamic discovery of web service descriptions. It basically defines how to use a WSDL description to generate the UDDI registration for a service.

First, divide the WSDL description into two parts (two separate WSDL files). The first file becomes the interface description. It includes the data types, messages, port types, and bindings. The second file is known as the implementation description. It includes only the service definition. The implementation description imports the interface description using the `<wsdl:import />` mechanism.

6.5.1 Interface Description

Example 6-21 is the interface description for our Hello World example.

Example 6-21. HelloWorldInterfaceDescription.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="HelloWorldInterfaceDescription"
   targetNamespace="urn:HelloWorldInterface"
   xmlns:tns="urn:HelloWorldInterface"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:message name="sayHello_IN">
    <part name="name" type="xsd:string" />
  </wsdl:message>
  <wsdl:message name="sayHello_Out">
    <part name="greeting" type="xsd:string" />
  </wsdl:message>

  <wsdl:portType name="HelloWorldInterface">
    <wsdl:operation name="sayHello">
      <wsdl:input message="tns:sayHello_IN" />
      <wsdl:output message="tns:sayHello_OUT" />
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="HelloWorldBinding"
                type="tns:HelloWorldInterface">
    <soap:binding style="rpc"
     transport="http://schemas.xmlsoap.org/soap/http"
    />
    <wsdl:operation name="sayHello">
      <soap:operation soapAction="urn:Hello" />
```

```
      <wsdl:input>
        <soap:body use="encoded"

        namespace="urn:Hello"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="encoded"
                  namespace="urn:Hello"

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
</wsdl:definitions>
```

6.5.2 Implementation Description

Example 6-22 is the WSDL implementation description for our Hello World example.

Example 6-22. HelloWorldImplementationDescription.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="HelloWorldImplementationDescription"
   targetNamespace="urn:HelloWorldImplementation"
   xmlns:tns="urn:HelloWorldImplementation"
   xmlns:hwi="urn:HelloWorldInterface"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

  <wsdl:import namespace="urn:HelloWorldInterface"
               location="HelloWorldInterfaceDescription.wsdl" />

  <wsdl:service name="HelloWorldService">
    <wsdl:port name="HelloWorldPort"
               binding="hwi:HelloWorldBinding">
      <!-- location of the Perl Hello World Service -->
      <soap:address
        location="http://localhost:8080" />
    </wsdl:port>
  </wsdl:service>

</wsdl:definitions>
```

6.5.3 Registering

Register the interface description as a UDDI `TModel`. You've seen NAICS categories and D-U-N-S identifiers as `TModel`s. Another type of `TModel` is a WSDL description of a service interface.

To register the interface as a `TModel`, create a `TModel` structure and use the `save_TModel` operation as in Example 6-23.

Example 6-23. Registering the interface description as a TModel

```
TModel TModel = new TModel(  );
TModel.setName("Hello World Interface");
```

The `OverviewDoc` is a pointer to the interface WSDL description, held on a publicly available web server. Example 6-24 shows how to set this.

Example 6-24. Setting the OverviewDoc

```
OverviewDoc odoc = new OverviewDoc(  );
// localhost == the name of the server where
// the WSDL can be accessed
odoc.setOverviewURL("http://localhost/HelloWorldInterface.wsdl");
TModel.setOverviewDoc(odoc);
```

Indicate that this `TModel` represents a WSDL interface description by creating a category reference with a `TModelKey` of `uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4`, a key name of `uddi-org:types`, and a key value of `wsdlSpec`, as shown in Example 6-25.

Example 6-25. Marking the TModel as WSDL

```
CategoryBag cbag = new CategoryBag(  );
KeyedReference kr = new KeyedReference(  );
kr.setTModelKey("uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4");
kr.setKeyName("uddi-org:types");
kr.setKeyValue("wsdlSpec");
```

Example 6-26 shows how to call the `save_TModel` operation to register the `TModel`.

Example 6-26. Calling the save_TModel operation

```
UDDIProxy proxy = new UDDIProxy(  );
proxy.setPublishURL(
"https://www-3.ibm.com/services/uddi/
                        testregistry/protect/publishapi");
AuthToken token = proxy.get_authToken("james", "semaj");
Vector TModels = new Vector(  );
TModels.add(TModel);
TModelDetail  detail  =  proxy.save_TModel(token.getAuthInfo().getText(    ),
TModels);
```

The `save_TModel` operation returns a copy of the `TModel` record just registered, including the automatically generated unique identifier. We keep that unique key for the next step, as shown in Example 6-27.

Example 6-27. Retaining unique key

```
TModel = (TModel)detail.getTModelVector(  ).elementAt(0);
String TModelKey = TModel.getTModelKey(  );
```

Now say what the service is and where it lives, as in Example 6-28.

Example 6-28. Defining a service, binding template, and access point for the service

```
BusinessService service = new BusinessService(  );
service.setBusinessKey(businessKey);
service.setName("HelloWorldService");

BindingTemplates templates = new BindingTemplates(  );
BindingTemplate template = new BindingTemplate(  );
```

```
templates.getBindingTemplateVector(  ).add(template);
service.setBindingTemplates(templates);

AccessPoint accessPoint = new AccessPoint(  );
accessPoint.setURLType("HTTP");
accessPoint.setText("http://localhost:8080");
template.setAccessPoint(accessPoint);
```

Example 6-29 specifies that this service is an instance of the `HelloWorld-InterfaceDescription TModel` just registered. The variable `TModelKey` is the unique identifier fetched in Example 6-27.

Example 6-29. Associate service with TModel

```
TModelInstanceDetails details = new TModelInstanceDetails(  );
TModelInstanceInfo instance = new TModelInstanceInfo(  );
instance.setTModelKey(TModelKey);
```

Provide a link to the WSDL implementation description as in Example 6-30. This, like the interface description, needs to be located at some publicly available web address.

Example 6-30. Linking to WSDL implementation description

```
InstanceDetails instanceDetails = new InstanceDetails(  );
OverviewDoc odoc = new OverviewDoc(  );
odoc.setOverviewURL("http://localhost/HelloWorldImplementationDescription.w
sdl");
instanceDetails.setOverviewDoc(odoc);
instance.setInstanceDetails(instanceDetails);
details.getTModelInstanceInfoVector(  ).add(instance);
template.setTModelInstanceDetails(details);
```

Once the registration is prepared, initialize the proxy and call the `save_service` operation to register the business service. Example 6-31 shows this, in abbreviated form. See Appendix C for the full source.

Example 6-31. Saving the service information

```
UDDIProxy proxy = new UDDIProxy(  );
// ...abbreviated
proxy.save_service(authInfo, services);
```

By following these guidelines, WSDL and UDDI can be made to work very well together.

6.6 Using UDDI and WSDL Together

Once the WSDL-defined web service is published into a UDDI registry, it is possible to build highly dynamic service proxies. The IBM Web Services ToolKit, for example, provides built-in support for locating services in UDDI and invoking those services through a dynamically configured WSDL-based proxy.

To show you more of what is going on behind the scenes, however, we're going to use UDDI4J and WSIF together to implement the same type of functionality.

The steps are simple:

1. Locate the Hello World service in the UDDI registry.
2. Access the WSDL description for the Hello World service.
3. Invoke the Hello World service.

All this is done on the client side. Nothing has to be done on the server for this to work.

First, write the code to locate the Hello World service in UDDI. Example 6-32 searches with `FindQualifiers` and takes the first result offered by the UDDI server.

Example 6-32. Locating a Hello World service

```
UDDIProxy proxy = new UDDIProxy(  );
FindQualifiers fq = new FindQualifiers(  );

ServiceList list = proxy.find_service(businessKey, "HelloWorldService", fq,
0);
ServiceInfos infos = list.getServiceInfos(  );

ServiceInfo      info      =      (ServiceInfo)infos.getServiceInfoVector(
).elementAt(0);
String serviceKey = info.getServiceKey(  );
```

With the unique identifier of the matching service, Example 6-33 goes to the UDDI registry to retrieve the business service. The binding template for that service then identifies the implementation to use.

Example 6-33. Locating an implementation

```
ServiceDetail detail = proxy.get_serviceDetail(serviceKey);
BusinessService service = (BusinessService)detail.
    getBusinessServiceVector(  ).elementAt(0);

BindingTemplate template = (BindingTemplate)service.
    getBindingTemplates().getBindingTemplateVector(  ).elementAt(0);
TModelInstanceDetails details = template.getTModelInstanceDetails(  );
TModelInstanceInfo    instance    =    details.getTModelInstanceInfoVector(
).elementAt(0);
InstanceDetails instanceDetails  = instance.getInstanceDetails(  );

OverviewDoc odoc = instanceDetails.getOverviewDoc(  );
String wsdlpath = odoc.getOverviewURLString(  );
```

WSDL in hand, Example 6-34 uses WSIF to invoke the web service.

Example 6-34. Invoking the web service with WSIF

```
Definition def = WSIFUtils.readWSDL(null, wsdlPath);
Service service = WSIFUtils.selectService(def, null, "HelloWorldService");
PortType      portType      =      WSIFUtils.selectPortType(def,      null,
"HelloWorldInterface");

WSIFDynamicPortFactory  dpf  =  new  WSIFDynamicPortFactory(def,  service,
portType);
WSIFPort port = dpf.getPort(  );
```

Typically, message creation is done behind the scenes, out of the sight of the programmer. Example 6-35 shows this.

Example 6-35. Creating messages with WSIF

```
WSIFMessage input = port.createInputMessage(  );
WSIFMessage output = port.createOutputMessage(  );
WSIFMessage fault = port.createFaultMessage(  );
```

Example 6-36 calls the Hello World service.

Example 6-36. Invoking Hello World

```
WSIFPart namePart = new WSIFJavaPart(String.class, args[0]);
input.setPart("name", namePart);

System.out.println("Calling the SOAP Server to say hello!\n");
System.out.print("The SOAP Server says: ");
port.executeRequestResponseOperation("sayHello", input, output, fault);

WSIFPart greetingPart = output.getPart("greeting");
String greeting = (String)greetingPart.getJavaValue(  );
System.out.print(greeting + "\n");
```

Running this produces the same output we saw in the other Hello World services examples (shown in Example 6-37).

Example 6-37. Output from the WSDL, UDDI, and WSIF Hello World client

```
C:\book>java wsdluddiExample James
Calling the SOAP Server to say hello!

The SOAP Server says: Hello James
```

The program dynamically discovered, inspected, and bound to the Hello World web services. We didn't program the client knowing which implementation we'd use. While the client was in Java, there's no reason we couldn't have written it in any language. C#, Visual Basic, and Perl all have UDDI and WSDL extensions.

6.7 The Web Service Inspection Language (WS-Inspection)

While UDDI is the best-known mechanism for service discovery, it is neither the only mechanism nor always the best tool for the job. In many cases, the complexity and scope of UDDI is overkill if all that is needed is a simple pointer to a WSDL document or a services URL endpoint. Recognizing this, IBM and Microsoft got together and worked out a proposal for a new Web Service Inspection Language that can be used to create a simple index of service descriptions at a given network location.

An example WS-Inspection document is illustrated in Example 6-38. It contains a reference to a single service (Hello World) with two descriptions—one WSDL-based description and one UDDI-based description.

Example 6-38. A simple WS-Inspection document

```
<?xml version="1.0"?>
<inspection
   xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
   xmlns:uddi="http://schemas.xmlsoap.org/ws/2001/10/inspection/uddi/">

<service>
 <abstract>The Hello World Service</abstract>

 <description
   referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
   location="http://example.com/helloworld.wsdl"/>

 <description referencedNamespace="urn:uddi-org:api">
   <uddi:serviceDescription
      location="http://www.example.com/uddi/inquiryapi">
   <uddi:serviceKey>
      4FA28580-5C39-11D5-9FCF-BB3200333F79
    </uddi:serviceKey>
   </uddi:serviceDescription>
  </description>
 </service>

<link
   referencedNamespace="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
   location="http://example.com/moreservices.wsil"/>

</inspection>
```

Once created, WS-Inspection documents should be placed in a well-known or easilydiscoverable location on your web server. In fact, the WS-Inspection specification defines that, at a minimum, an inspection document called *Inspection.wsil* should be available at the root the server: for instance, http://www.ibm.com/inspection.wsil. This allows potential clients of those services to locate inspection documents easily and thereby discover the services being advertised.

The relationship between UDDI and WS-Inspection is simple. UDDI is a phone book. If you need a plumber to fix the pipes under your kitchen sink but do not know of a good one to call, you open the phone book and find one. If you need a web service that implements a particular WSDL defined port type for processing purchase orders for ball bearings, you can submit a request to a UDDI registry to find an appropriate service. WS-Inspection, however, is useful if you already know the service provider you want to use (e.g., you already know which plumber who want to call so you dont have to look in the phonebook). You'd simply refer to the WS-Inspection document published by the service provider to find the location of the services they are offering.

6.7.1 WS-Inspection Syntax

The syntax of a WS-Inspection document is simple. The root `inspection` element contains a collection of `abstract`, `link`, and `service` elements. The `abstract` element provides for simple documentation throughout the WS-Inspection document. The `link` element allows the inspection document to link to other external inspection documents or even other discovery mechanisms (such as a UDDI registry) where additional information can be found. The

`service` element represents a web service being offered by the publisher of the inspection document.

The `service` element itself is a collection of `abstract` and `description` elements. You can describe a service in several ways. WS-Inspection allows all a service's descriptions to be listed. You can provide extended information about each service description using XML extensibility. Example 6-38, for instance, contains both a WSDL and UDDI-based description.

WS-Inspection will be submitted for standardization at some point. For now, both IBM and Microsoft have implemented support for it in their web services offerings and other web service toolkit vendors are considering doing the same. Because of its usefulness and simple syntax, WS-Inspection is likely to develop favorable support.